CHALMERS
UNIVERSITY OF TECHNOLOGY

Supporting verification-driven incremental distributed design of components

Supporting Verification-Driven Incremental Distributed Design of Components

Claudio Menghi[1(✉)], Paola Spoletini[2], Marsha Chechik[3], and Carlo Ghezzi[4]

[1] Chalmers | University of Gothenburg, Gothenburg, Sweden
claudio.menghi@gu.se
[2] Kennesaw State University, Marietta, USA
pspoleti@kennesaw.edu
[3] University of Toronto, Toronto, Canada
chechik@cs.toronto.edu
[4] Politecnico di Milano, Milan, Italy
carlo.ghezzi@polimi.it

Abstract. Software systems are usually formed by multiple components which interact with one another. In large systems, components themselves can be complex systems that need to be decomposed into multiple sub-components. Hence, system design must follow a systematic approach, based on a recursive decomposition strategy. This paper proposes a comprehensive verification-driven framework which provides support for designers during development. The framework supports hierarchical decomposition of components into sub-components through formal specification in terms of pre- and post-conditions as well as independent development, reuse and verification of sub-components.

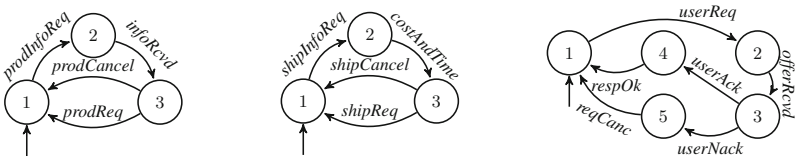
1 Introduction

Software is usually not a monolithic product: it is often comprised of multiple components that interact with each other to provide the desired functionality. Components themselves can be complex, requiring their own decomposition into sub-components. Hence, system design, must follow a systematic approach, based on a recursive decomposition strategy that yields a modular structure. A good decomposition and a careful specification should allow components and sub-components to be developed in isolation by different development teams, delegated to third parties [32], or reused off-the-shelf.

In this context, guaranteeing correctness of the system under development becomes particularly challenging because of the intrinsic tension between two main requirements. On the one hand, to handle complexity, we need to enable development of sub-components where only a partial view of the system is available [28]. On the other hand, we must ensure that independently developed and verified (sub-)components can be composed to guarantee global correctness of

A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 169–188, 2018.
https://doi.org/10.1007/978-3-319-89363-1_10

The p&d running example. The p&d system supports furniture purchase and delivery. It uses two existing web services, which implement furniture-sale and delivery, as well as a component that implements the user interface. These are modeled by the labeled transition systems shown in Fig. 1a-1c. The p&d component under design is responsible for interaction with these components, which form its execution environment. The overall system must ensure satisfaction of the properties informally described in Fig. 1d.



(a) Furniture-sale. (b) Shipping. (c) User.

P1: ship and product info are provided only if a request has been received.
P2: when user requests are processed, offers are considered only after users received information about the desired product.
P3: the furniture service is activated only if the user has decided to purchase.
P4: when a user request is cancelled by the p&d system, no user ack precedes the cancellation.

(d) Properties of the p&d system.

Fig. 1. The p&d running example.

the resulting system. Thus, we believe that component development should be supported by a process that (1) is intrinsically iterative; (2) supports decentralized development; and (3) guarantees correctness at each development stage.

The need for supporting incremental development of components has been widely recognized. Some approaches [15,37] synthesize a partial model of components from properties and scenarios and facilitate an iterative development of this model through refinement. Others [7,8,10,26,27] provide support for checking and refining partial models, with the goal of preserving correctness when such systems get refined. However, while these techniques guarantee correctness at each development stage, they do not address the problem of decentralized development.

In this paper, we describe a unified framework called FIDDle (a Framework for Iterative and Distributed Design of components) which supports decentralized top-down development. FIDDle supports a formal specification of global properties, a decomposition process and specification of component interfaces by providing a set of tools to guarantee correctness of the different artifacts produced during the process. The main contribution of the paper is a method for supporting an iterative and distributed verification-driven component development process through a coherent set of tools. Specific novel contributions are (1) a new formalism, called *Interface Partial Labelled Transition System (IPLTS)*, for specifying components through a decomposition that encapsulates sub-components into unspecified black-box states; (2) an approach to specify *the expected behavior of black-box states* via pre- and post-conditions expressed in Fluent Linear Time Temporal Logic; and (3) a notion of *component correctness*

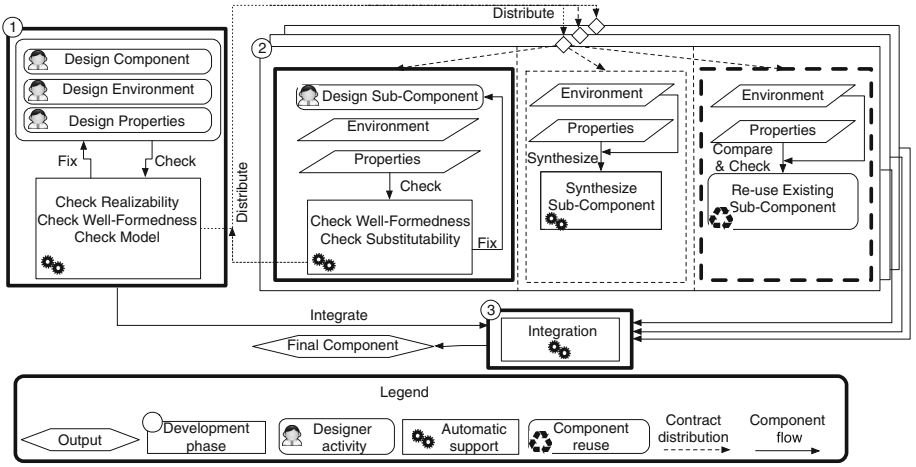


Fig. 2. Overview of the application of FIDDle for developing a component. Thick-bordered components are implemented in FIDDle. Thick-dashed bordered components are currently supported by the theory presented in this paper, but they are still not fully implemented. Thin-dashed bordered components are not discussed in this work.

and a *local verification procedure* that *guarantees preservation of global properties* once the components are composed.

We illustrate FIDDle using a simple example: the *purchase&delivery* (p&d) example [14,29] – see Fig. 1. We evaluate FIDDle on a realistic case study obtained by reverse-engineering the executive module of the Mars Rover developed at NASA [12,17,18]. Scalability is evaluated by considering randomly-generated examples.

Organization. Sect. 2 provides an overview of FIDDle. Section 3 gives the necessary background. Section 4 presents Interface Partial Labelled Transition Systems (IPLTS). Section 5 defines a set of algorithms for reasoning on partial components and describes their implementation. Section 6 reports on an evaluation of the proposed approach. Section 7 compares FIDDle with related approaches, and Sect. 8 concludes. Proofs for the theorems in the paper can be found in the Appendix available at http://ksuweb.kennesaw.edu/~pspoleti/fase-appendix.pdf; source code and video of the tool and a complete replication package can be found at https://github.com/claudiomenghi/FIDDLE.

2 Overview

FIDDle is a verification-driven environment supporting incremental and distributed component development. A high-level view of FIDDle is shown in Fig. 2. FIDDle allows incrementally developing a component through a set of development phases in which the human insight and experience are exploited (rounded boxes labeled with a designer icon or a recycle symbol, to indicate design or reuse,

respectively) and phases in which automated support is provided (squared boxes labeled with a pair of gearwheels). Automatic support allows verifying the current state of the design, synthesizing parts of the partial component, or checking whether the designed sub-component can correctly fit into the original design. FIDDle development phases are described below.

Creating an Initial Component Design. This phase is identified in Fig. 2 with the symbol ①. The development team formalizes the properties that this component has to guarantee and designs an initial, high-level structure of the component. Designers also formulate properties that the component needs to ensure. The initial component design is created using a state-based formalism that can clearly identify parts (called "sub-components" in this paper), represented as *black-box* states, whose internal design is delayed to a later stage or split apart for distributed development by other parties. In the following, we refer to other states as "regular". Black-box states are enriched with an *interface* that provides information on the universe of events relevant to the black-box. They are also decorated with via pre- and post-conditions that allow distributed teams to develop sub-components without the need to know about the rest of the system. The *contract* of a black box state consists of its interface and pre- and post-conditions.

In the p&d example, the environment (assumed as given) in which the p&d component will be deployed is composed by the furniture-sale component (Fig. 1a), the shipping component (Fig. 1b) and the user (Fig. 1c). A possible initial design for the p&d component is shown in Fig. 3c. It contains the regular states 1 and 3 and black-box states 2 and 4. The initial state is state 1. Whenever a *userReq* event is detected, the component moves from the initial state 1 into the black-box state 2, which represents a sub-component in charge of managing the user request. An event *offerRcvd* which indicates that an offer is provided to the user labels the transition to state 3. The pre- and post- conditions for black-box states 2 and 4 are shown in Fig. 3b. Events *prodInfoReq*, *infoRcvd*, *shipInfoReq* and *costAndTime* can occur while the component is in the black-box state 2. The pre-condition requires that there is a user request that has not yet been handled, while the post-condition ensures that the furniture-sale and the shipping services provided info on the product and on delivery cost and time. FIDDle supports the developer in checking properties of the initial component design.

The *realizability checker* confirms the existence of an integration that completes the partially specified component and ensures the satisfaction of the properties of interest. If such a component does not exist, the designer needs to redesign the partially-specified component. The *well-formedness checker* verifies that both the pre- and the post-conditions of black-box states are satisfiable. Finally, the *model checker* verifies whether the (partial) component (together with its contract) guarantees satisfaction of the properties of interest.

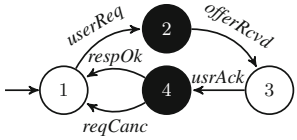
In the p&d example, the model checker identifies a problem with the partial solution sketched in Fig. 3c. No matter how the black-box state 2 is to be defined, the p&d component cannot satisfy property *P4* since every time *reqCanc* occurs

$P1 = (\neg((\neg F\_UserReq)\,\mathcal{U}\,(F\_ShipInfoReq \vee F\_ProdInfoReq)))$
$P2 = \Box(F\_UserReq \rightarrow (\neg((\neg F\_InfoRcvd)\,\mathcal{U}\,F\_OfferRcvd)))$
$P3 = \Box(F\_UsrReq \rightarrow ((\neg((\neg F\_UserAck)\,\mathcal{W}\,F\_ShipReq))$
$P4 = \Box((F\_UsrReq \wedge ((\neg F\_UsrReq)\,\mathcal{U}\,F\_ReqCanc)) \rightarrow ((\neg F\_UserAck)\,\mathcal{U}\,F\_ReqCanc))$

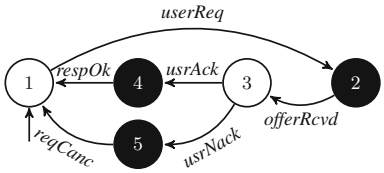
(a) FLTL formulation of the p&d properties.

	State 2
interface	{ *prodInfoReq*, *infoRcvd*, *shipInfoReq*, *costAndTime* }
pre	$\Diamond(F\_UserReq \wedge \neg\Diamond(F\_RespOk \vee F\_ReqCanc))$
post	$(\Diamond F\_InfoRcvd) \wedge (\Diamond F\_CostAndTime)$
	State 4
interface	{ *prodReq*, *shipReq* }
pre	$\Box(F\_UserReq \rightarrow \Diamond F\_InfoRcvd)$
post	$(\Diamond(F\_ProdReq) \wedge \Diamond(F\_ShipReq))$
	State 5
interface	{ *prodCancel*, *shipCancel* }
pre	$\Box(F\_UserReq \rightarrow \Diamond F\_InfoRcvd)$
post	$(\Diamond(F\_ProdCancel) \wedge \Diamond(F\_ShipCancel))$

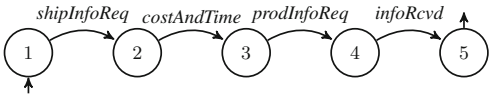
(b) Contracts for black-box states of Figs. 3c-3g.



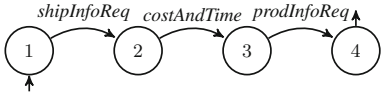
(c) Partial p&d.



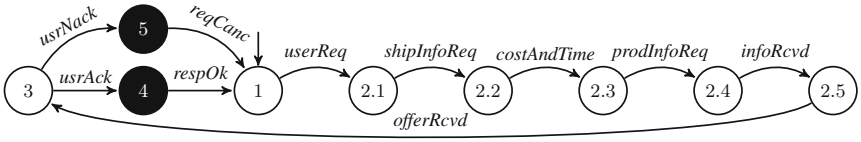
(d) Another partial p&d component.



(e) A sub-component for black-box state 2.



(f) Another sub-component for black-box state 2.



(g) Integration of the sub-component of Fig. 3e and the component of Fig. 3d.

Fig. 3. The p&d running example: artifacts produced by FIDDle.

it is preceded by *usrAck*. This suggests a re-design of the p&d component, which may lead to a new model, shown in Fig. 3d. This model includes two regular states: state 1, in which the component waits for a new user request, and state 3, in which the component has provided the user with an offer and is waiting for an answer. The user might accept (*userAck*) or reject (*userNack*) an offer and, depending on this choice, either state 4 or 5 is entered. States 2, 4 and 5 are black-box states, to be refined later. The designer also provides pre- and post-conditions for the black-box states. Pre- and post-conditions of the black-box state 2 specify that there is a pending user request, and that cost, time and product information are collected. Pre- and post-conditions of the black-box state 4 specify that *infoRcvd* has occurred after the user request, and both a

product and shipping requests are performed. Finally, pre- and post-conditions of the black-box state 5 specify that *infoRcvd* has occurred after the user request and before entering the state, and both the product and the shipping requests are cancelled when leaving the state. This model is checked using the provided tools; since it passes all the checks, it can be used in the next phase of the development.

The design team may choose to refine the component or *distribute* the development of unspecified sub-components (represented by black box states) to other (internal or external) development teams. In both cases, the sub-component can be designed by only considering the contract of the corresponding black-box state. Each team can develop the assigned sub-component or reuse existing components.

Sub-component Development. This phase is identified in Fig. 2 with the symbol ②. Each team can design the assigned sub-component using any available technique, including manual design (left side), reusing of existing sub-components (right side) or synthesizing new ones from the provided specifications (center). The only constraints are (1) given the stated pre-condition, the sub-component has to satisfy its post-condition, and (2) the sub-component should operate in the same environment as the overall partially specified component. Sub-component development can itself be an iterative process, but neither the model of the environment nor the overall properties of the system can be changed during this process. Otherwise, the resulting sub-component cannot be automatically integrated into the overall system.

In the p&d example, development of the sub-component for the black-box state 2 is delegated to an external contractor. Candidate sub-components are shown in Fig. 3e–f. In the former case, the component requests shipping info details and waits until the shipping service provides the shipment cost and time. Then it queries the furniture-sale service to obtain the product info. In the latter case, the shipping and the furniture services are queried, but the sub-component does not wait for an answer from the furniture-sale. Since these candidates are fully defined, the well-formedness check is not needed. Yet, the *substitutability checking* confirms that of these, only the sub-component in Fig. 3e satisfies the post-condition in Fig. 3b.

Sub-component Integration. This phase is identified in Fig. 2 with the symbol ③. FIDDle guarantees that if each sub-component is developed correctly w.r.t. the contract of the corresponding black-box state, the component obtained by integrating the sub-components is also correct. In the p&d example, the sub-component in Fig. 3e passes the substitutability check and can be a valid implementation of the black-box state 2 in Fig. 3d. Their integration is showed in Fig. 3g.

3 Preliminaries

The model of the environment and the properties of interest are expressed using Labelled Transition Systems and Fluent Linear Time Temporal Logic.

Model of the Environment. Let *Act* be the universal set of observable events and let $Act_\tau = Act \cup \{\tau\}$, where τ denotes an unobservable local event. A *Labeled Transition System (LTS)* [20] is a tuple $A = \langle Q, q_0, \alpha A, \Delta \rangle$, where Q is the set of states, $q_0 \in Q$ is the initial state, $\alpha A \subseteq Act$ is a finite set of events, and $\Delta \subseteq Q \times \alpha A \cup \{\tau\} \times Q$ is the transition relation. The parallel composition operation is defined as usual (see for example [14]).

Properties. A fluent [33] Fl is a tuple $\langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, where $I_{Fl} \subset Act$, $T_{Fl} \subset Act$, $I_{Fl} \cap T_{Fl} = \emptyset$ and $Init_{Fl} \in \{true, false\}$. A fluent may be *true* or *false*. A fluent is *true* if it has been initialized by an event $i \in I_{Fl}$ at an earlier time point (or if it was initially *true*, that is, $Init_{Fl} = true$) and has not yet been terminated by another event $t \in T_{Fl}$; otherwise, it is *false*. For example, consider the LTS in Fig. 1c and the fluent $F\_ReqPend = \langle \{userReq\}, \{respOk,\ reqCanc\},\ false \rangle$. *F_ReqPend* holds in a trace of the LTS from the moment at which *userReq* occurs and until a transition labeled with *respOk* or *reqCanc* is fired. In the following, we use the notation *F_Event* to indicate a fluent that is *true* when the event with label *event* occurs.

An FLTL formula is obtained by composing fluents with standard LTL operators: $\bigcirc$ (next), $\Diamond$ (eventually), $\Box$ (always), $\mathcal{U}$ (until) and $\mathcal{W}$ (weak until). For example, FLTL encodings of the properties *P1*, *P2*, *P3* and *P4* are shown in Fig. 3a.

Satisfaction of FLTL formulae can be evaluated over *finite* and *infinite* traces, by first constructing and FLTL interpretation of the infinite and finite trace and then by evaluating the FLTL formulae over this interpretation The FLTL interpretation of a finite trace is obtained by slightly changing the interpretation of infinite traces. The evaluation of the FLTL formulae on the finite trace is obtained by considering the standard interpretation of LTL operator over finite traces (see [13]). In the following, we assume that Definitions 5 and 4 (available in the Appendix) are considered to evaluate whether an FLTL formula is satisfied on finite and infinite traces, respectively.

4 Modeling and Refining Components

This section introduces a novel formalism for modeling and refining components. We define the notion of a partial LTS and then extend it with pre- and post-conditions.

Partial LTS. A *partial LTS* is an LTS where some states are "regular" and others are "black-box". Black-box states model portions of the component whose behavior still has to be specified. Each black-box state is augmented with an interface that specifies the universe of events that can occur in the black-box. A *Partial LTS (PLTS)* is a structure $P = \langle A, R, B, \sigma \rangle$, where: $A = \langle Q, q_0, \alpha A, \Delta \rangle$ is an LTS; Q is the set of states, s.t. $Q = R \cup B$ and $R \cap B = \emptyset$; R is the set of *regular* states; B is the set of *black-box* states; $\sigma : B \rightarrow 2^{\alpha A}$ is the *interface*. An LTS is a PLTS where the set of black-box states is empty. The PLTS in Fig. 3d is defined over the regular states 1 and 3, and the black-box states 2,

4 and 5. The interface specifies that events *prodInfoReq*, *infoRcvd*, *shipInfoReq* and *costAndTime* can occur in the black-box state 2.

Definition 1. *Given a PLTS* $P = \langle A, R, B, \sigma \rangle$ *defined over the LTS* $A = \langle Q^A, q_0^A, \alpha A, \Delta^A \rangle$ *and an LTS* $D = \langle Q^D, q_0^D, \alpha D, \Delta^D \rangle$*, the parallel composition* $P \parallel D$ *is an LTS* $S = \langle Q^S, q_0^S, \alpha S, \Delta^S \rangle$ *such that* $Q^S = Q^A \times Q^D$*;* $q_0^S = (q_0^A, q_0^D)$*;* $\alpha S = \alpha A \cup \alpha D$*; and the set of transitions* Δ^S *is defined as follows:*

- $\frac{(s,l,s') \in \Delta^A}{(\langle s,t \rangle, l, \langle s',t \rangle) \in \Delta^S}$*, and* $l \in \alpha A \setminus \alpha D$ *or* $l = \tau$*;*
- $\frac{(t,l,t') \in \Delta^D}{(\langle s,t \rangle, l, \langle s,t' \rangle) \in \Delta^S}$*, and one of the following is satisfied: (1)* $l \in \alpha D \setminus \alpha A$*, (2)* $l = \tau$*, or (3)* $(s \in B$ *and* $l \in \sigma(s))$*;*
- $\frac{(s,l,s') \in \Delta^A, (t,l,t') \in \Delta^D}{(\langle s,t \rangle, l, \langle s',t' \rangle) \in \Delta^S}$ *and* $l \in \alpha A \cap \alpha D, l \neq \tau$*.*

Given P, A, D defined above, the system $S = P \parallel D$ and a state q of P, we say that a finite trace $l_0, l_1, \ldots l_n$ of S *reaches* q if there exists a sequence $\langle s_0, t_0 \rangle, l_0, \langle s_1, t_1 \rangle, \ldots l_n, \langle q, t_{n+1} \rangle$, where for every $0 \leq i \leq n$, we have $(\langle s_i, t_i \rangle, l_i, \langle s_{i+1}, t_{i+1} \rangle) \in \Delta^S$. For example, considering the PLTS in Fig. 3d and the LTS in Fig. 1c, the finite trace obtained by performing a *userReq* event reaches the black-box state 2 of the PLTS.

Given a finite trace $\pi = l_0, l_1, \ldots l_n$ (or an infinite trace $l_0, l_1, \ldots$) of S, we say that its sub-trace $l_i, l_{i+1} \ldots l_k$ is *inside* the black-box state b if one of the sub-sequences associated with π is in the form $\langle b, t_i \rangle, l_i, \langle b, t_{i+1} \rangle, \ldots, l_k, \langle b, t_k \rangle$, where $l_i, l_{i+1}, \ldots, l_k \in \sigma(b)$. Note that a sub-trace is a *finite* trace. For example, considering the parallel composition of the PLTS in Fig. 3d and the LTSs in Fig. 1c and b, and the finite trace associated with events *userReq, shipInfoReq, offerRcvd*, the sub-trace associated with *shipInfoReq* is inside the black-box state 2. This means that *shipInfoReq* must occur in the sub-component replacing the black-box state 2.

Adding Pre- and Post-conditions. The intended behavior of a sub-component refining a black-box state can be captured using pre- and post-conditions. The *contract* for the sub-component associated with a box consists of the box interface and its pre- and post-conditions. Given the universal set *FLTL* of the FLTL formulae, an *Interface PLTS* (IPLTS) I is a structure $\langle A, R, B, \sigma, pre, post \rangle$, where $\langle A, R, B, \sigma \rangle$ is a PLTS, $pre : B \rightarrow FLTL$ and $post : B \rightarrow FLTL$.

For each black-box state b, the function *pre* specifies a constraint that must be satisfied by all *finite* traces of P that reach b. For example, the FLTL-expressed pre-condition for the black-box state 4 of the IPLTS in Fig. 3d requires that any trace of the composition between the IPLTS and an LTS that reaches the black-box state 4 provides info on the product to the user after his/her request.

For each black-box state b, the function *post* specifies *a post-condition* that constrains the behavior of the system in any sub-trace performed inside b. For example, the post-condition of the black-box state 4 of the IPLTS in Fig. 3d ensures that whenever this IPLTS is composed with an LTS, a product request and a shipping request are performed by the furniture-sale service while the system is inside the black-box state.

Given an IPLTS I and an LTS D, the *parallel composition* S between I and D is obtained by considering the PLTS P associated with I and the LTS D as specified in Definition 1. Given an IPLTS I, an LTS D and the *parallel composition* S between I and D, trace π of S is *valid* iff it is infinite and for every black-box state b, the post-condition $post(b)$ holds in any sub-trace of π performed inside b.

Definition 2. *Given an LTS D, an IPLTS I is* well-formed *(over D) iff every valid trace of $S = I \parallel D$ satisfies all the pre-conditions of black-box states of I.*

We say that $S = I \parallel D$ *satisfies* an FLTL property ϕ if and only if ϕ is satisfied by every valid trace of S. In the p&d example, the post-condition $\Diamond$ (*F_ProdReq*) $\wedge \Diamond$ (*F_ShipReq*) of the black-box 4 ensures that the parallel composition of the component in Fig. 3d and its environment satisfies *P3*.

Sub-components and Their Integration. Integration aims to replace black-box states of a given IPLTS with the corresponding sub-components. Given an IPLTS I, one of its black-box states b and its interface $\sigma(b)$, *a sub-component for* b is an IPLTS R defined over the set of events $\sigma(b)$. One state q_f^R of R is defined as the *final state* of R. Given a sub-component R, an LTS of its environment E, and a trace in the form $\pi_i;\pi_e$ such that $\pi_i = l_0, l_1 \ldots l_n$ and $\pi_e = l_{n+1}, l_{n+2}, \ldots l_k$, we say that $\pi_i;\pi_e$ is *a trace of the parallel composition between R and E* if and only if (1) there exists a sequence $q_0, l_0, q_1, l_1 \ldots l_n, q_n$ in the environment such that for all i, where $0 \leq i < n$, (q_i, l_i, q_{i+1}) is a transition of E; (2) π_e is obtained by $R \parallel E$ considering q_n as the initial state for the environment, (3) π_e reaches q_f^R. A sub-component is *valid* if it ensures that the traces of the parallel composition satisfy its post-conditions. Intuitively, a trace of the parallel composition between a sub-component R and the environment E is obtained by concatenating two sub-traces: π_i and π_e. The sub-trace π_i corresponds to a set of transitions performed by the environment before the sub-component is activated, while π_e is a trace the system generates while it is in the sub-component R.

Definition 3. *Given an IPLTS I with a black-box state b, the environment E and a sub-component R for b, R is a* substitutable sub-component *iff every trace $\pi_i;\pi_e$ of the parallel composition between R and E is such that if π_i satisfies pre(b) then π_e guarantees post(b).*

Intuitively, whenever the sub-component is entered and the pre-condition *pre(b)* is satisfied (i.e., the trace π_i satisfies *pre(b)*), then a trace of the parallel composition between the sub-component and the environment that reaches the final state of the sub-component must satisfy the post-condition *post(b)*.

A black-box state of an IPLTS C can be replaced by a substitutable sub-component R though an integration procedure. The resulting IPLTS C' is called *integration*. Intuitively, the integration procedure connects every incoming and outgoing transition of the considered black-box state to the initial and final state of the substitutable sub-component R, respectively. Integrating the sub-component R for black-box state 2 in Fig. 3e into the component in Fig. 3d produces the IPLTS in Fig. 3g. The prefix "2." is used to identify the states

obtained from R. The contracts of black-box states 4 and 5 are the same as those in Fig. 3b.

Theorem 1. *Given a well-formed IPLTS C and a substitutable sub-component R for a black-box state b of C, if C satisfies an FLTL property ϕ, then the integration C' obtained by substituting b with R also satisfies ϕ.*

The sub-component R from Fig. 3e is substitutable; thus, integrating it into the partial component C shown in Fig. 3g ensures that the resulting integrated component C' preserves properties *P1-P4*.

5 Verification Algorithms

In this section, we describe the algorithms for the analysis of partial components, which we have implemented on top of LTSA [25].

Checking Realizability. Realizability of a property ϕ is checked via the following procedure. Let E be the environment of the partial component C, and C^B be the LTS resulting from removing all black-box states and their incoming and outgoing transitions from C. Check $C^B \parallel E \models \phi$. If ϕ is not satisfied, the component is not realizable: no matter how the black-box states are specified, there will be a behavior of the system that does not satisfy ϕ. Otherwise, compute $C \parallel E$ (as specified in Definition 1) and model-check it against $\neg\phi$. If the property $\neg\phi$ is satisfied, the component is not realizable. Indeed, all the behaviors of $C \parallel E$ satisfy $\neg\phi$, i.e., there is no behavior that the component can exhibit to satisfy ϕ. Otherwise, the component may be realizable. For example, the realizability checker shows that it is possible to realize a component refining the one shown in Fig. 3c while satisfying property *P2*. Specifically, it returns a trace that ensures that after a *userReq* event, the offer is provided to the user (the event *offerRcvd*) only if the furniture service has confirmed the availability of the requested product (the event *inforRcvd*).

Theorem 2. *Given a component specified using an IPLTS C, its environment E, and a property of interest ϕ, the realizability checker returns "not realizable" if there is no component C' obtained from C by integrating sub-components, s.t. $(C' \parallel E) \models \phi$.*

Checking Well-Formedness. Given a partial component C with a black-box state b annotated with a pre-condition *pre(b)* and its environment E, the well-formedness checks whether *pre(b)* is satisfied in C as follows.

(1) *Transform post-conditions into LTSs.* Transform every FLTL post-condition $\text{post}(b_i)$ of every black-box state b_i of C, including b, into an FLTL formula $\text{post}(b_i)'$ as specified in [13]. This transformation ensures that the *infinite* traces that satisfy $\text{post}(b_i)'$ have the form $\pi, \{end\}^\omega$, where π satisfies $\text{post}(b_i)$. For each black-box state b_i, the corresponding post-condition

post$(b_i)'$ is transformed into an equivalent LTS, called LTS_{b_i}, using the procedure in [37]. Since LTS_{b_i} has traces in the form $\pi, \{end\}^\omega$, it has a state s with an *end*-labelled self-loop. This self-loop is removed, and s is considered as final state of LTS_{b_i}. All other *end*-labeled transitions are replaced by τ-transitions. Each automaton LTS_{b_i} contains all the traces that do not violate the corresponding post-condition.

(2) *Integrate the LTSs of all the black-box states* $b_i \neq b$. For every black-box state $b_i \neq b$, eliminate b_i and add LTS_{b_i} to C by replacing every incoming transition of b_i with a transition whose destination is the initial state of LTS_{b_i}, and every outgoing transition of b_i with a transition whose source is the final state of LTS_{b_i}. This step creates an LTS which encodes all the traces of the component that do not violate any post-conditions of its black-box states.

(3) *Integrate the LTS of the black-box state b.* Integrate LTS_b into C together with two additional states, q_1 and q_2, calling the resulting model C'. Replace every incoming transition of b by a transition with destination q_1. Replace every outgoing transition of b by a transition whose source is the final state of LTS$_b$. Add a transition labeled with τ from q_1 to the initial state of LTS$_b$. Add a self-loop labeled with an event *end* to q_2. Add a τ-transition from q_1 to q_2. The obtained LTS C' encodes all the valid traces of the system. When a valid trace reaches the black-box state b, C' can enter state q_2 from which only the *end*-labelled self-loop is available.

(4) *Verify.* Recall that the precondition *pre(b)* of b is defined over finite traces, i.e., those that reach the initial state of the sub-component to be substituted for b. To use standard verification procedures, we transform *pre(b)* into an equivalent formula, *pre(b)'*, over infinite traces. This transformation, specified in [13], ensures that every trace of the form $\pi, \{end\}^\omega$ satisfies *pre(b)'* iff π satisfies *pre(b)*. By construction in step 3 above, $C' \parallel E$ has a valid trace of this form which is generated when $C \parallel E$ reaches the initial state of the LTS LTS_b associated with the black-box state b of C. To check the pre-condition, we verify whether $C' \parallel E \models pre(b)'$ using traditional model checking.

In the p&d example, if we remove the clause $\Diamond$ *F_InfoRcvd* from the post-condition of the black-box state 2, the p&d component is not well-formed since the pre-condition of state 4 is violated. The counterexample shows a trace that reaches the black-box state 4 in which an event *userReq* is not followed by *infoRcvd*. Adding $\Diamond$ *F_InfoRcvd* to the post-condition of state 2 solves the problem.

Theorem 3. *Given a partial component C with a black-box state b annotated with a pre-condition pre(b) and its environment E, the well-formedness procedure returns true iff the valid traces of C satisfy the pre-condition pre(b).*

Model Checking. To check whether $C \parallel E$ satisfies ϕ, we first construct an LTS C' that generates only valid traces, by plugging into C the LTSs corresponding to all of its black-box states (as done in steps 1 and 2 of the well-formedness check) and use a classical FLTL model-checker to verify $C' \parallel E \models \phi$. If we consider the

design of Fig. 3d and assume that the black-box state 2 is not associated with any post-condition, the model checker returns the counterexample *userReq*,τ, *offerRcvd* for property *P2*, since the sub-component that will replace the black-box state 2 is not forced to ask to book the furniture service. Adding the post-condition in Fig. 3b solves the problem.

Theorem 4. *The model checking procedure returns* true *iff every valid trace of* $C \parallel E$ *satisfies* ϕ.

Checking Substitutability. Given the environment E, a component C with a black-box state b and pre- and post-conditions *pre(b)* and *post(b)*, and a sub-component R, this procedure checks whether R can be used in C in place of b. We first present a procedure assuming that R has no black-box states.

(1) *Transform the pre-condition pre(b) into an LTS, called* LTS_b, using Step (1) of the well-formedness procedure.
(2) *Compute the sequential composition* $(LTS_b.R)$ *between the* LTS_b *and* R. This is done by connecting the final state q_1 of LTS_b with the initial state of R by a transition labelled with a fresh event *init*. Then, the final state of R is connected to an additional state q_2 through a τ-labeled transition. A self-loop labeled with a fresh event *end* is added to q_2. Performing these steps ensures that the prefix π of every infinite trace in the form $\pi, \{end\}^\omega$ is comprised of two parts: $\pi = \pi_1; \pi_2$, where π_1 satisfies *pre(b)* and π_2 is generated by the LTS R.
(3) *Verify the result.* The formula $\lambda = init \rightarrow \bigcirc(post(b))$ must hold on any trace that reaches the final state of R, e.g., on any trace of the form $\pi; \{end\}^\omega$, where λ' is the result of applying the finite- to infinite-trace FLTL transformation [13] to λ. This transformation ensures that π satisfies λ iff a trace of the form $\pi; \{end\}^\omega$ satisfies λ'. And that, in turn, can be verified by checking $((LTS_b.R) \parallel E) \models \lambda'$ using a classical model-checker.

If R contains black-box states, checking R requires performing Steps (1) and (2) of the well-formedness check before running the substitutability procedure.

In the p&d example, the substitutability checker does not return any counterexample for the sub-component in Fig. 3e. Thus, the post-condition is satisfied and the sub-component can be integrated in place of the black-box state 2.

Theorem 5. *Let a component C with a black-box state b, its pre- and post-conditions pre(b) and post(b), a sub-component R, and C's environment E be given. The substitutability checker returns true, indicating that R can be used in C in place of b, iff for every trace $\pi = \pi_i; \pi_e$ of $R \parallel E$, if π_i is the finite prefix of E satisfying pre(b) and π_e is obtained by $R \parallel E$ considering the final state of π_i as the initial state of the environment, then π_e satisfies post(b).*

6 Evaluation

We aim to answer two questions: **RQ.1**: How effective is FIDDle w.r.t. supporting an iterative, distributed development of correct components? (Sect. 6.1) and **RQ.2**: How scalable is the automated part of the proposed approach? (Sect. 6.2).

6.1 Assessing Effectiveness

We simulated development of a complex component and analyzed FIDDle-provided support along the steps described in Sect. 2.

Experimental Setup. We chose the executive module of the K9 Mars Rover developed at NASA Ames [12,17,18] and specified using LTSs. The overall size of the LTS is $\sim 10^7$ states. The executive module was made by several components: *Executive*, *ExecCondChecker*, *ActionExecution* and *Database*. *ExecCondChecker* was further decomposed into *db-monitor* and *internal*. Each of these components was associated with a shared variable (*exec*, *conditionList*, *action* and *db*, respectively) used to communicate with the other components, e.g., the *exec* variable was used by *ExecCondChecker* to communicate with *Executive*. The access of each shared variable was regulated through a condition variable and a lock. The complete model of the *Executive* component comprised of 11 states, each further decomposed as an LTS. The final model of the *Executive* component was obtained by replacing these states with the corresponding LTSs. This model had about 100 states which is a realistic component of a medium size [5,6,24].

We considered two properties: ($\mathcal{P}1$): *Executive* performed an action only after a new plan was read from *Database*; ($\mathcal{P}2$): *Executive* got the lock over the *condList* variable only after obtaining the *exec* lock.

Creating an Initial Component Design. We considered the existing model (*D3*) of the *Executive* and abstracted portions of the complete model into black-box states to create two partial components *D1* and *D2* representing partial designs. To generate *D2* we encapsulated three states that receive plans and prepare for plan execution into the black-box state *Read_Plans*. To generate *D1*, we also set one of the 10 states of the *Executive* whose corresponding LTS is in charge of executing a plan, i.e., state *ExecuteTaskAction*, as a black-box state. By following this procedure, *D3* and *D2* can be obtained from *D2* and *D1*, respectively, by integrating the abstracted sub-components.

We considered the (partial) components *D1*, *D2* and *D3* and used FIDDle to iteratively develop and check their contracts. For *D1*, the steps were as follows: (1) The *realizability checker* confirmed the existence of a model that refined *D1* and satisfied the properties of interest. (2) The *model checker* returned a counterexample for both properties of interest. For $\mathcal{P}1$, the model checker returned a counterexample in which no plan was read and yet an action was performed. For $\mathcal{P}2$, the counterexample was where *Executive* got the *condList* lock without possessing the *exec* lock. To guarantee the satisfaction of $\mathcal{P}1$, we specified a post-condition to the black-box state *Read_Plans* that ensures that a plan was read. We also added a pre-condition requiring that an action was not under execution when the black-box state *Read_Plans* was entered. (3) The *well-formedness checker* returned a counterexample trace that reached the black-box state *Read_Plans* while an action was under execution. (4) To ensure well-formedness, we added a postcondition to the black-box state *ExecuteTaskAction* ensuring that an action was not under execution when the system exited the black-box state. (5) The *model checker* confirmed that $\mathcal{P}1$ held. (6) To guar-

antee the satisfaction of $\mathcal{P}2$, we added a post-condition to the black-box state *Read_Plans* ensuring that when the control left the black-box, $\mathcal{P}2$ remained *true* and the *Executive* had the *exec* lock.

For design *D2*, the steps were as follows: (1) The *realizability checker* confirmed the existence of a model that refined *D2* and satisfied the properties of interest. (2) We ran the model checker that returned a counterexample for both properties of interest. (3) We added to the black-box state *Read_Plans* the same pre- and post-conditions of as those developed for design *D1* and ran the *well-formedness* and the *model checker*. (4) The *well-formedness checker* confirmed that *D2* satisfied the pre-condition of the black-box *Read_Plans*; the *model checker* certified the satisfaction of $\mathcal{P}1$ and $\mathcal{P}2$.

Since the model of *Executive* was complete, we ran only the *model checker* to check *D3*. Properties $\mathcal{P}1$ and $\mathcal{P}2$ were satisfied.

Sub-component Development. We simulated a refinement process in which pre- and post-conditions were given to third parties for sub-component development. We considered the sub-components *SUB1* and *SUB2* containing the portion of the *Executive* component abstracted by the black-box states *ExecuteTaskAction* and *Read_Plans*, respectively. We run the *substitutability checker* to verify, affirmatively, whether *SUB1* and *SUB2* ensured the post-condition of the black-box states *ExecuteTaskAction* and *Read_Plans* given their pre-conditions.

Sub-component Integration. We then plugged in the designed sub-components into their corresponding black-box states. We integrated each sub-component into design *D1* and used the *model checker* to verify the resulting (partial) components w.r.t. properties $\mathcal{P}1$-$\mathcal{P}2$. The properties were satisfied, as intended.

Results. FIDDle was effective in analyzing partial components and helping change their design to ensure the satisfaction of the properties of interest. The experiment confirmed the possibility of distributing the design of sub-components for the black-box states. As expected, no rework at the integration level was required, i.e., integration produced components that satisfied the properties of interest. This confirmed that FIDDle supports verification-driven iterative and distributed development of components.

Threats to Validity. A threat to construct validity concerns the (manual) construction of intermediate model produced during development by abstracting an existing component model and the design of the properties to be considered. However, the intermediate partial designs and the selected properties were based on original developer comments present in the model. A threat to internal validity concerns the design of the contracts (pre- and post- conditions and interfaces) for the black-box states chosen along the process. However, pre- and post- conditions were chosen and designed by consulting property specification patterns proposed in literature [16]. The fact that a single example has been considered is a threat to external validity. However, the considered example is a medium-size complex real case study [6,22,35].

Table 1. Results of experiments $E1$ and $E2$.

	#CompStates													
	$E1 : (T_w)/(T_m)$							$E2 : (T_s)/(T_m)$						
#EnvStates	10	50	100	250	500	750	1000	10	50	100	250	500	750	1000
10	1.45	1.26	1.51	1.29	1.42	1.43	1.31	2.20	4.37	2.18	1.50	2.19	1.62	1.62
100	1.15	1.25	1.50	1.08	0.88	1.02	2.33	3.51	4.66	3.61	2.80	3.18	1.96	2.73
1000	1.39	1.23	0.60	1.44	4.90	1.00	2.83	13.98	8.12	3.84	2.64	2.83	2.91	2.00

6.2 Assessing Scalability

We set up two experiments (*E1* and *E2*) comparing performance of the *well-formedness* and the *substitutability checkers* w.r.t. classical model checking as the size of the partial components under development and their environments grew. Our experiments were based on a set of *randomly-generated* models.

E1. To evaluate the *well-formedness checker*, we generated an LTS model of the environment and a complete model for the component. We checked the parallel composition between the component and the environment w.r.t. a property of interest using a standard model checker. Then, we generated a partial component by marking one of the states of the complete component as a black-box, defining pre- and post- conditions for it and ran the well-formedness checker, comparing performance of the two.

E2. To evaluate the *substitutability checker*, we generated a complete component as in the previous experiment. Then, we extracted a sub-component by selecting half of the component states and the transitions between them. States q_0 and q_f were added to the sub-component as the initial and final state, respectively. State q_0 (q_f) was connected with all the states of the sub-component that had, in the original component, at least one incoming (resp., outgoing) transition from (resp., to) a state that was not added to the sub-component. We defined the pre- and post-conditions for the sub-component and ran the substitutability checker comparing its performance with model-checking.

Experimental Setup. We implemented a *random model generator* to create LTSs with a specified number of states, transition density (transitions per state) and number of events. We generated environments with an increasing number of states: 10, 100 and 1000. We have chosen 10 as a fixed value for the transition density and 50 as the cardinality of the set of events. We considered components with 10, 50, 100, 250, 500, 750 and 1000 states. The components were generated using the same transition density and number of events as in the produced environment. To produce the partial component, we considered one of the states of the component obtained previously as a black-box, and randomly selected 25% of the events of the component as the interface of the partial component. To produce the sub-component, we randomly extracted half of the component states and the transitions between them.

Properties of Interest, Pre- and Post-conditions. We considered properties $\mathcal{K}1 = \Box(Q \rightarrow P)$, $\mathcal{K}2 = \Diamond Q \rightarrow (\neg P \,\mathcal{U}\, Q)$, $\mathcal{K}3 = \Box(Q \rightarrow \Box(\neg P))$, which correspond to commonly used property patterns [16], and where Q and P are appropriately defined fluents. We considered $\mathcal{K}1$, $\mathcal{K}2$ and $\mathcal{K}3$ as pre- and post-conditions for the black-box.

Methodology and Results. We ran each experiment 5 times on a 2 GHz Intel Core i7, with 8 GB 1600 MHz DDR3 disk. For each combination of values of the #EnvStates and #ContStates we computed the average between the time required by the well-formedness checker (T_w) and by the model checker (T_m), for the experiment $E1$, and the average between the time required by the substitutability checker (T_s) and by the model checker (T_m), for the experiment $E2$ (see Table 1). The results show that the well-formedness and the substitutability checker scale as the classical model checker.

Threats to Validity. The procedure employed to randomly generate models is a threat to construct validity. However, the transition density of the components was chosen based on the Mars Rover example. Furthermore, the number of states of the sub-component was chosen such that the ratio between the sizes of the component and the sub-component was approximately the same of the Mars Rover. The properties considered in the experiment are a threat to internal validity. However, they were chosen by consulting property specification patterns proposed in literature [16]. Considering a single black-box state is a threat to external validity. However, our goal was to evaluate how FIDDle scales with respect to the component and the environment sizes and not w.r.t. the number of black-box states and the size of the post-conditions. Considering multiple black-box states reduces to the case of considering a single black-box with a more complex post-condition.

7 Related Work

We discuss approaches for developing incrementally correct components.

Modeling Partiality. Modal Transition Systems [21], Partial Kripke Structures [8], and LTS$^{\uparrow}$ [17] support the specification of incomplete concurrent systems and can be used in an iterative development process. Other formalisms, such as Hierarchical State Machines (HSMs) [4], are used to model sequential processes via a top-down development process but can only be analyzed when a fully-specified model is available.

Checking Partial Models. Approaches to analyze partial models (e.g., [8,10]) are not applicable to the problem considered in this paper where missing sub-components are specified using contracts and their development is distributed across different development teams. The assumption generation problem for LTSs [17] is complementary to the one considered in this paper and concerns the computation of an assumption that describes how the system model interacts with the environment.

Substitutability Checking. The goal of substitutability checking is to verify whether a possibly partial sub-component can be plugged into a higher level structure without affecting its correctness. Problems such as "compositional reasoning" [1,19,30], "component substitutability" [9], and "hierarchical model checking" [4] are related to this part of our work. Our work differs because we first guarantee that the properties of interest are satisfied in the initially-defined partial component and then check that the provided sub-components can be plugged into the initial component.

Synthesis. Program synthesis [14,31] aims at computing a model of the system that satisfies the properties of interest. Moreover, synthesis can be used to generate assumptions on a system's environment to make its specification relizable (e.g., [23]). Sketch [36] supports programmers in describing an initial structure of the program that can be completed using synthesis techniques, but does not explicitly consider models. Many techniques for synthesizing components have been proposed, e.g., [14,37], and a fully automated synthesis of highly non-trivial components of over 2000 states big is becoming possible [11] for special cases, by limiting the types of synthesizable goals and using heuristics. However, such cases might not be applicable in general. Recent work has been done in the direction of compositional [2,3] and distributed [34] synthesis. We do not consider our approach to be an alternative to synthesis, but instead a way to combine synthesis techniques with the human design.

8 Conclusion

We presented a verification-driven methodology, called FIDDle, to support iterative distributed development of components. It enables recursively decomposing a component into a set of sub-components so that the correctness of the overall component is ensured. Development of sub-components that satisfy their specifications can then be done independently, via distributed development. We have evaluated FIDDle on a realistic Mars Rover case study. Scalability was evaluated using randomly generated examples.

Acknowledgments. Research partly supported from the EU H2020 Research and Innovation Programme under GA No. 731869 (Co4Robots).

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Meth. Softw. Des. **15**(1), 7–48 (1999)
2. Alur, R., Moarref, S., Topcu, U.: Pattern-Based Refinement of Assume-Guarantee Specifications in Reactive Synthesis. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 501–516. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_49

3. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
4. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM SIGSOFT Softw. Eng. Notes **23**(6), 175–188 (1998)
5. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Reverse engineering finite state machines from rich internet applications. In: Proceedings of the 15th Working Conference on Reverse Engineering, pp. 69–73 (2008)
6. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing conformance of real-time applications by automatic generation of observer. In: Proceedings of RV, Electronic Notes in Theoretical Computer Science, pp. 23–43 (2004)
7. Bernasconi, A., Menghi, C., Spoletini, P., Zuck, L.D., Ghezzi, C.: From model checking to a temporal proof for partial models. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 54–69. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_4
8. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_25
9. Chaki, S., Clarke, E.M., Sharygina, N., Sinha, N.: Verification of evolving software via component substitutability analysis. Formal Methods Softw. Des. **32**(3), 235–266 (2008)
10. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. **12**(4), 371–408 (2003)
11. Ciolek, D., Braberman, V.A., D'Ippolito, N., Uchitel, S.: Technical Report: Directed Controller Synthesis of Discrete Event Systems. CoRR, abs/1605.09772 (2016)
12. Cobleigh, J.M., Giannakopoulou, D., PĂsĂreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_24
13. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness. In: Proceedings of AAAI, pp. 1027–1033 (2014)
14. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, U.: Synthesising non-anomalous event-based controllers for liveness goals. ACM Tran. Softw. Eng. Methodol. **22**, 9 (2013)
15. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Controllability in partial and uncertain environments. In: Proceedings of ACSD, pp. 52–61. IEEE (2014)
16. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of FMSP, pp. 7–15. ACM (1998)
17. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proceedings of ASE, pp. 3–12. IEEE (2002)
18. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. J. Autom. Softw. Eng. **12**(3), 297–320 (2005)
19. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
20. Keller, R.M.: Formal verification of parallel programs. Commun. ACM **19**(7), 371–384 (1976)
21. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proceedings of LICS, pp. 203–210. IEEE (1988)

22. Levy, L.S.: Taming the Tiger: Software Engineering and Software Economics. Springer Books on Professional Computing Series. Springer-Verlag, New York (1987). https://doi.org/10.1007/978-1-4612-4718-0
23. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: Proceedings of ACM/IEEE MEMPCODE, pp. 43–50 (2011)
24. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of ICSE, pp. 501–510 (2008)
25. Magee, J., Kramer, J.: State Models and Java Programs. Wiley, New York (1999)
26. Menghi, C., Spoletini, P., Ghezzi, C.: Dealing with incompleteness in automata-based model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 531–550. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_32
27. Menghi, C., Spoletini, P., Ghezzi, C.: Integrating goal model analysis with iterative design. In: Grünbacher, P., Perini, A. (eds.) REFSQ 2017. LNCS, vol. 10153, pp. 112–128. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54045-0_9
28. Nivoit, J.-B.: Issues in strategic management of large-scale software product line development. Master's thesis, MIT, USA (2013)
29. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: Bussler, C., Fensel, D. (eds.) AIMSA 2004. LNCS (LNAI), vol. 3192, pp. 106–115. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30106-6_11
30. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series, pp. 123–144. Springer-Verlag, New York Inc (1985). https://doi.org/10.1007/978-3-642-82453-1_5
31. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of POPL, pp. 179–190. ACM (1989)
32. Pretschner, A., Broy, M., Kruger, I.H., Stauner, T.: Software engineering for automotive systems: a roadmap. In: Proceedings of FOSE, pp. 55–71. IEEE Computer Society (2007)
33. Sandewall, E.: Features and Fluents (Vol. 1): The Representation of Knowledge about Dynamical Systems. Oxford University Press Inc, New York (1995)
34. Sibay, G.E., Uchitel, S., Braberman, V., Kramer, J.: Distribution of modal transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 403–417. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_33
35. Software Measurement Services Ltd. "small project", "medium-size project", and "large project": What do these terms mean? (2004). http://www.totalmetrics.com/function-points-downloads/Function-Point-Scale-Project-Size.pdf
36. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis. University of California, Berkeley (2008)
37. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Trans.Softw. Eng. **35**(3), 384–406 (2009)